

***SuperLU: Sparse Direct Solver and
Preconditioner***

X. Sherry Li

xqli@lbl.gov

<http://crd.lbl.gov/~xiaoye/SuperLU>

***12th* DOE ACTS Collection Workshop**

August 16-19, 2011

- **Supports from DOE, NSF, DARPA**
 - **TOPS (Towards Optimal Petascale Simulations)**
 - **CEMM (Center for Extended MHD Modeling)**
- **Developers and contributors**
 - **Sherry Li, LBNL**
 - **James Demmel, UC Berkeley**
 - **John Gilbert, UC Santa Barbara**
 - **Laura Grigori, INRIA, France**
 - **Meiyue Shao, Umeå University, Sweden**
 - **Pietro Cicotti, UC San Diego**
 - **Daniel Schreiber, UIUC**
 - **Yu Wang, U. North Carolina, Charlotte**
 - **Ichitaro Yamazaki, LBNL**
 - **Eric Zhang, Albany High**

- **Download site** <http://crd.lbl.gov/~xiaoye/SuperLU>
 - **Users' Guide, HTML code documentation**
- **Gunzip, untar**
- **Follow README at top level directory**
 - **Edit make.inc for your platform (compilers, optimizations, libraries, ...)**
(may move to autoconf in the future)
 - **Link with a fast BLAS library**
 - The one under CBLAS/ is functional, but not optimized
 - Vendor, GotoBLAS, ATLAS, ...

Outline of Tutorial



- **Functionality**
- **Background of the algorithms**
 - **Differences between sequential and parallel solvers**
- **Sparse matrix data structure, distribution, and user interface**
- **Examples, Fortran 90 interface**

- Solving a system of linear equations $Ax = b$
 - Sparse: many zeros in A ; worth special treatment
- Iterative methods
 - A is not changed (read-only)
 - Key kernel: sparse matrix-vector multiply
 - Easier to optimize and parallelize
 - Low algorithmic complexity, but may not converge
- Direct methods
 - A is modified (factorized)
 - Harder to optimize and parallelize
 - Numerically robust, but higher algorithmic complexity
- Often use direct method to precondition iterative method

▪ Survey of different types of factorization codes

<http://crd.lbl.gov/~xiaoye/SuperLU/SparseDirectSurvey.pdf>

- LL^T (s.p.d.)
- LDL^T (symmetric indefinite)
- LU (nonsymmetric)
- QR (least squares)
- Sequential, shared-memory (multicore), distributed-memory, out-of-core
 - GPU, FPGA become active, no public code yet.
- Distributed-memory codes: usually MPI-based
 - SuperLU_DIST [Li/Demmel/Grigori/Yamazaki]
 - accessible from PETSc, Trilinos, . . .
 - MUMPS, PasTiX, WSMP, . . .

- LU decomposition, triangular solution
- Incomplete LU (ILU) preconditioner (serial SuperLU 4.0 up)
- Transposed system, multiple RHS
- Sparsity-preserving ordering
 - Minimum degree ordering applied to $A^T A$ or $A^T + A$ [MMD, Liu `85]
 - ‘Nested-dissection’ applied to $A^T A$ or $A^T + A$ [(Par)Metis, (PT)-Scotch]
- User-controllable pivoting
 - Pre-assigned row and/or column permutations
 - Partial pivoting with threshold
- Equilibration: $D_r A D_c$
- Condition number estimation
- Iterative refinement
- Componentwise error bounds [Skeel `79, Arioli/Demmel/Duff `89]

	SuperLU	SuperLU_MT	SuperLU_DIST
Platform	Serial	SMP, multicore	Distributed memory
Language	C	C + Pthreads or OpenMP	C + MPI
Data type	Real/complex, Single/double	Real/complex, Single/double	Real/complex, Double

- Fortran interfaces
- SuperLU_MT similar to SuperLU both numerically and in usage

■ Industry

- Cray Scientific Libraries
- FEMLAB
- HP Mathematical Library
- IMSL Numerical Library
- NAG
- Sun Performance Library
- Python (NumPy, SciPy)

■ Research

- In ACTS Tools: Hypre, PETSc, Overture, Trilinos
- M3D-C¹, NIMROD (burning plasmas for fusion energys)
- Omega3P (accelerator design)
- ...

Review of Gaussian Elimination (GE)



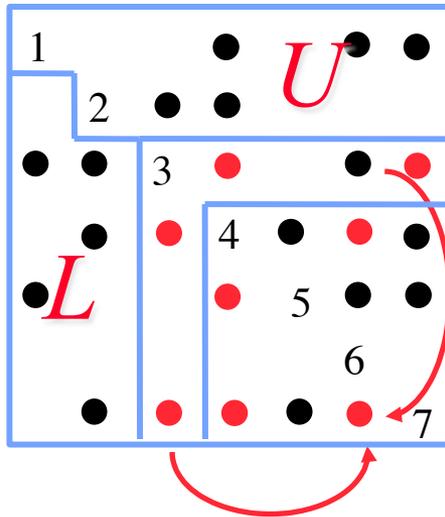
- Solving a system of linear equations $Ax = b$
- First step of GE: (make sure α not too small . . . Otherwise do pivoting)

$$A = \begin{bmatrix} \alpha & w^T \\ v & B \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ v/\alpha & I \end{bmatrix} \cdot \begin{bmatrix} \alpha & w^T \\ 0 & C \end{bmatrix}$$
$$C = B - \frac{v \cdot w^T}{\alpha}$$

- Repeats GE on C
- Results in $\{L \setminus U\}$ decomposition ($A = LU$)
 - L lower triangular with unit diagonal, U upper triangular
- Then, x is obtained by solving two triangular systems with L and U

■ Scalar algorithm: 3 nested loops

- Can re-arrange loops to get different variants: left-looking, right-looking, ...



```
for i = 1 to n
  column_scale ( A(:,i) )
  for k = i+1 to n s.t. A(i,k) != 0
    for j = i+1 to n s.t. A(j,i) != 0
      A(j,k) = A(j,k) - A(j,i) * A(i,k)
```

Typical fill-ratio: 10x for 2D problems, 30-50x for 3D problems

Data structure: Compressed Row Storage (CRS)



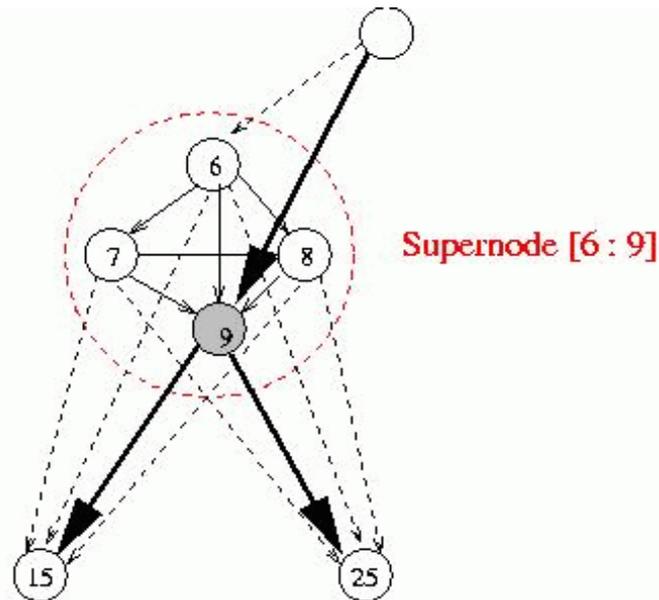
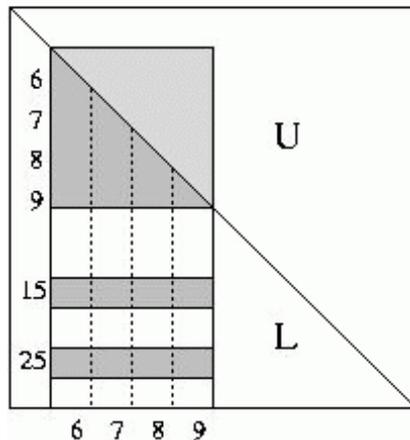
- Store nonzeros row by row contiguously
- Example: $N = 7$, $NNZ = 19$
- 3 arrays:
 - Storage: NNZ reals, $NNZ+N+1$ integers

$$\begin{pmatrix} 1 & & & & & & a \\ & 2 & & & & & b \\ c & d & 3 & & & & \\ & e & & 4 & & & f \\ & & & & 5 & & g \\ & & & h & i & 6 & j \\ & k & & l & & & 7 \end{pmatrix}$$

	1	3	5	8	11	13	17	20
nzval	1 a	2 b	c d 3	e 4 f	5 g	h i 6 j	k l 7	
colind	1 4	2 5	1 2 3	2 4 5	5 7	4 5 6 7	3 5 7	
rowptr	1 3 5 8 11 13 17 20							

Many other data structures: “Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods”, R. Barrett et al.

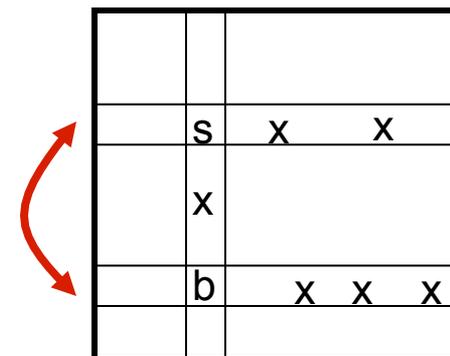
- Use (blocked) CRS or CCS, and any ordering method
 - Leave room for fill-ins ! (symbolic factorization)
- Exploit “supernode” (dense) structures in the factors
 - Can use Level 3 BLAS
 - Reduce inefficient indirect addressing (scatter/gather)
 - Reduce graph traversal time using a coarser graph



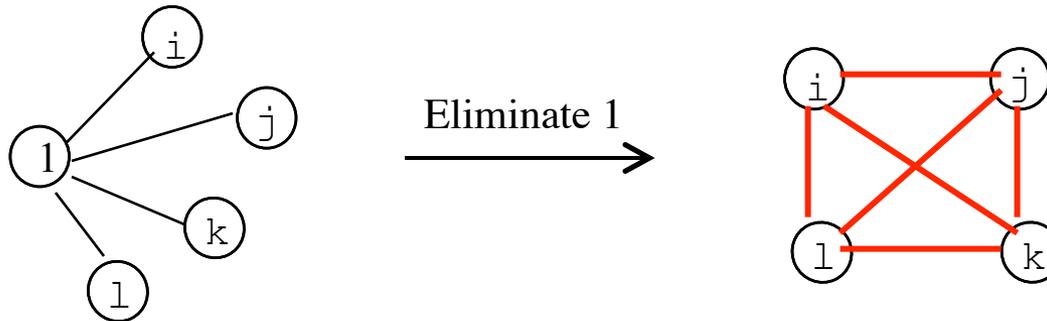
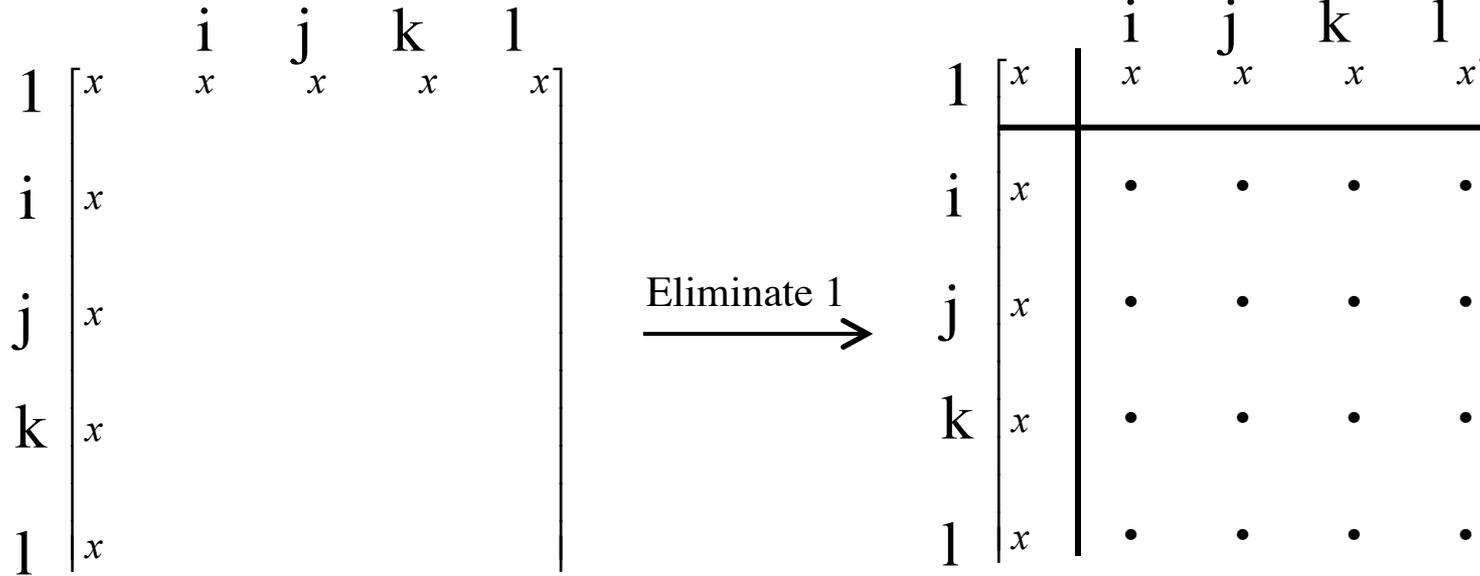
- **Sparse LU factorization: $P_r A P_c^T = L U$**
 - Choose permutations P_r and P_c for numerical stability, minimizing fill-in, and maximizing parallelism.

- **Phases for sparse direct solvers**
 1. **Order equations & variables to minimize fill-in.**
 - NP-hard, so use heuristics based on combinatorics.
 2. **Symbolic factorization.**
 - Identify supernodes, set up data structures and allocate memory for L & U.
 3. **Numerical factorization – usually dominates total time.**
 - How to pivot?
 4. **Triangular solutions – usually less than 5% total time.**

- Goal of pivoting is to control element growth in L & U for stability
 - For sparse factorizations, often relax the pivoting rule to trade with better sparsity and parallelism (e.g., threshold pivoting, static pivoting, ...)
- **Partial pivoting** used in sequential SuperLU and SuperLU_MT (GEPP)
 - Can force diagonal pivoting (controlled by diagonal threshold)
 - Hard to implement scalably for sparse factorization
- **Static pivoting** used in SuperLU_DIST (GESP)
 - Before factor, scale and permute A to maximize diagonal: $P_r D_r A D_c = A'$
 - During factor $A' = LU$, replace tiny pivots by $\sqrt{\varepsilon} \|A\|$, without changing data structures for L & U
 - If needed, use a few steps of iterative refinement after the first solution
 - ➔ quite stable in practice

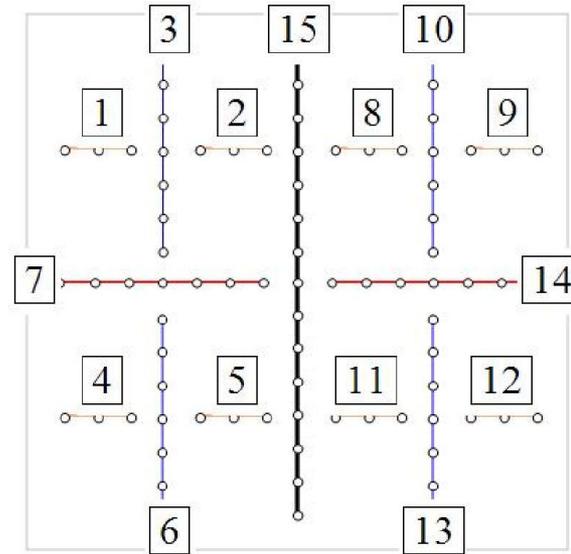
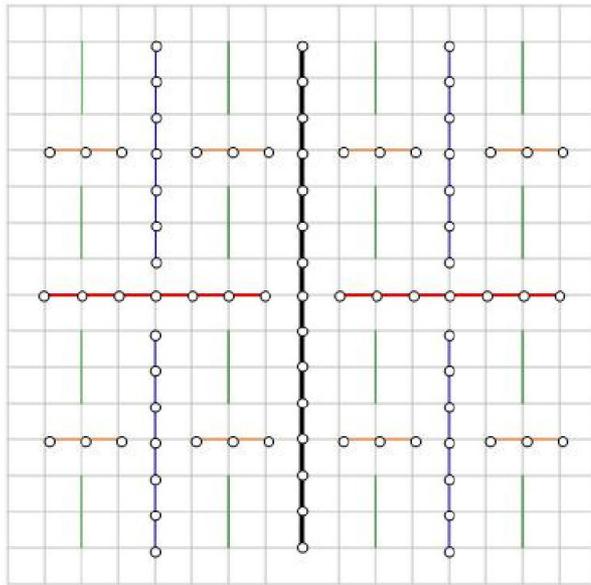


Local greedy: minimize upper bound on fill-in

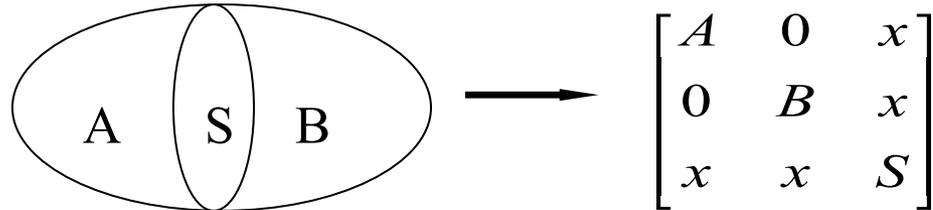


Ordering : Nested Dissection

- **Model problem: discretized system $Ax = b$ from certain PDEs, e.g., 5-point stencil on $n \times n$ grid, $N = n^2$**
 - **Factorization flops: $O(n^3) = O(N^{3/2})$**
- **Theorem: ND ordering gives optimal complexity in exact arithmetic [George '73, Hoffman/Martin/Rose]**

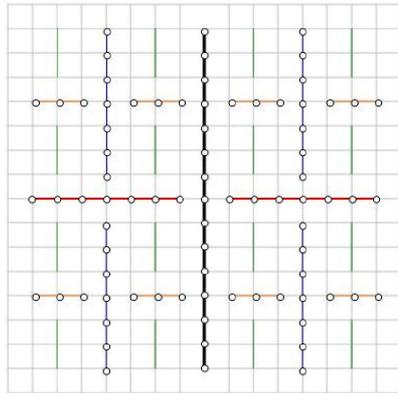


- **Generalized nested dissection [Lipton/Rose/Tarjan '79]**
 - **Global graph partitioning: top-down, divide-and-conquer**
 - **Best for largest problems**
 - **Parallel codes available: ParMetis, Scotch**
 - **First level**

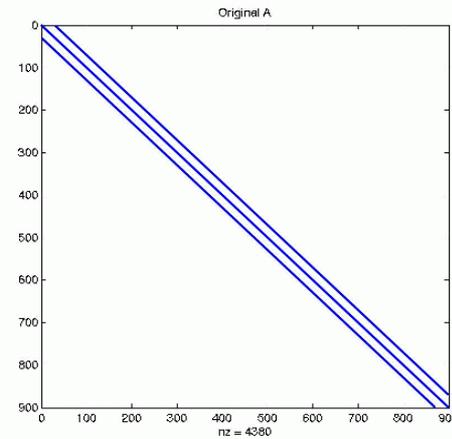


- **Recurse on A and B**
- **Goal: find the smallest possible separator S at each level**
 - **Multilevel schemes:**
 - **Chaco [Hendrickson/Leland '94], Metis [Karypis/Kumar '95]**
 - **Spectral bisection [Simon et al. '90-'95]**
 - **Geometric and spectral bisection [Chan/Gilbert/Teng '94]**

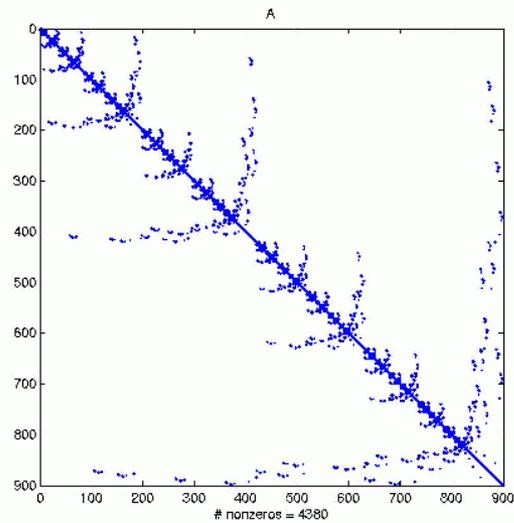
ND Ordering



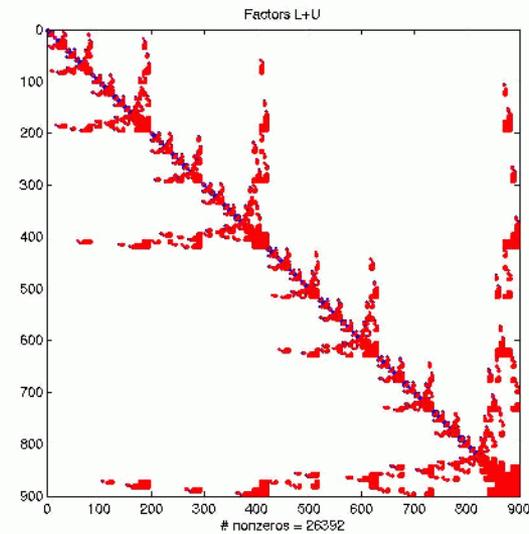
2D mesh



A, with row-wise ordering



A, with ND ordering



L & U factors

- Can use a symmetric ordering on a symmetrized matrix
 - Case of partial pivoting (serial SuperLU, SuperLU_MT):
Use ordering based on $A^T * A$
 - Case of static pivoting (SuperLU_DIST):
Use ordering based on $A^T + A$

- Can find better ordering based solely on A, without symmetrization
 - Diagonal Markowitz [Amestoy/Li/Ng `06]
 - Similar to minimum degree, but without symmetrization
 - Hypergraph partition [Boman, Grigori, et al. `08]
 - Similar to ND on $A^T A$, but no need to compute $A^T A$

- Library contains the following routines:
 - Ordering algorithms: MMD [J. Liu], COLAMD [T. Davis]
 - Utility routines: form A^T+A , $A^T A$
- Users may input any other permutation vector (e.g., using Metis, Chaco, etc.)

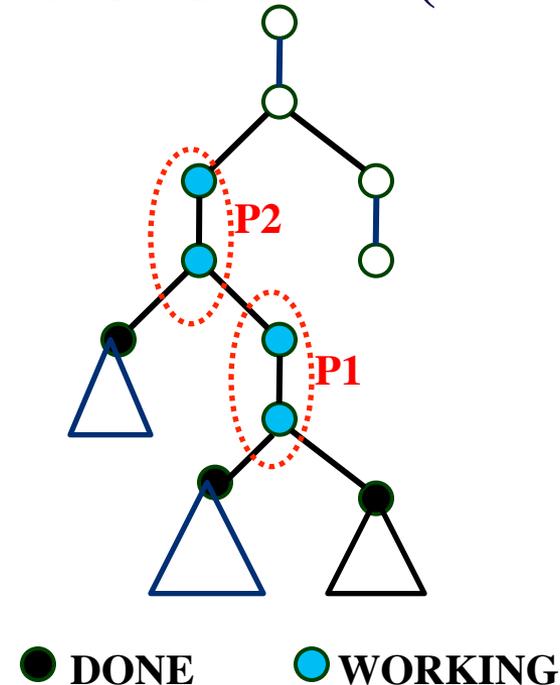
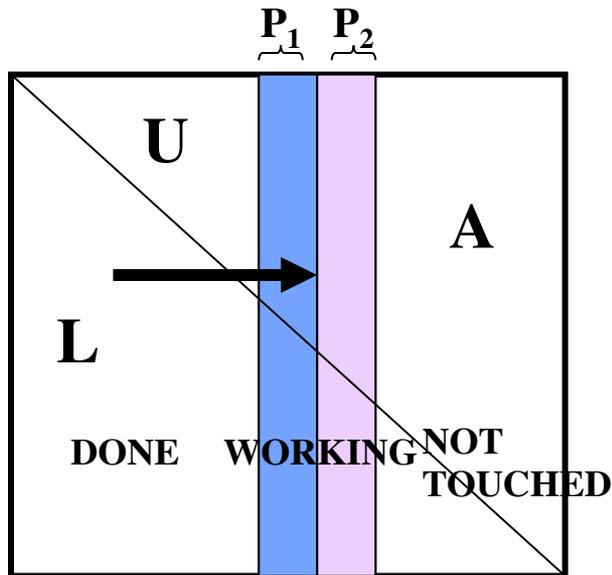
```
...  
set_default_options_dist ( &options );  
options.ColPerm = MY_PERMC; // modify default option  
ScalePermstructInit ( m, n, &ScalePermstruct );  
METIS ( ..., &ScalePermstruct.perm_c );  
...  
pdgssvx ( &options, ..., &ScalePermstruct, ... );  
...
```

- **Cholesky** [George/Liu `81 book]
 - Use **elimination graph of L and its transitive reduction (elimination tree)**
 - **Complexity linear in output: $O(\text{nnz}(L))$**

- **LU**
 - Use **elimination graphs of L & U and their transitive reductions (elimination DAGs)** [Tarjan/Rose `78, Gilbert/Liu `93, Gilbert `94]
 - **Improved by symmetric structure pruning** [Eisenstat/Liu `92]
 - **Improved by supernodes**
 - **Complexity greater than $\text{nnz}(L+U)$, but much smaller than $\text{flops}(LU)$**

- **Sequential SuperLU**
 - **Enhance data reuse in memory hierarchy by calling Level 3 BLAS on the supernodes**
- **SuperLU_MT**
 - **Exploit both coarse and fine grain parallelism**
 - **Employ dynamic scheduling to minimize parallel runtime**
- **SuperLU_DIST**
 - **Enhance scalability by static pivoting and 2D matrix distribution**

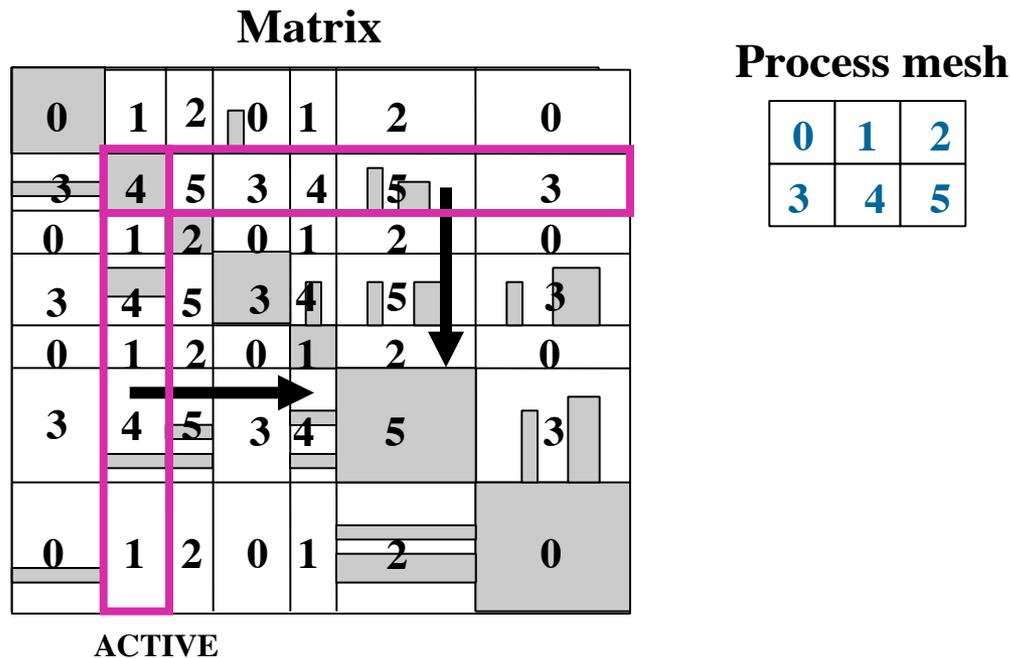
- Pthread or OpenMP
- **Left-looking** – relatively more READs than WRITEs
- Use shared task queue to schedule ready columns in the elimination tree (bottom up)
- Over 12x speedup on conventional 16-CPU SMPs (1999)



SuperLU_DIST [Li/Demmel/Grigori/Yamazaki]



- MPI
- **Right-looking** – relatively more WRITES than READS
- 2D block cyclic layout
- Look-ahead to overlap comm. & comp.
- Scales to 1000s processors



❖ Intel Clovertown:

- 2.33 GHz Xeon, 9.3 Gflops/core
- 2 sockets x 4 cores/socket
- L2 cache: 4 MB/2 cores

❖ Sun VictoriaFalls:

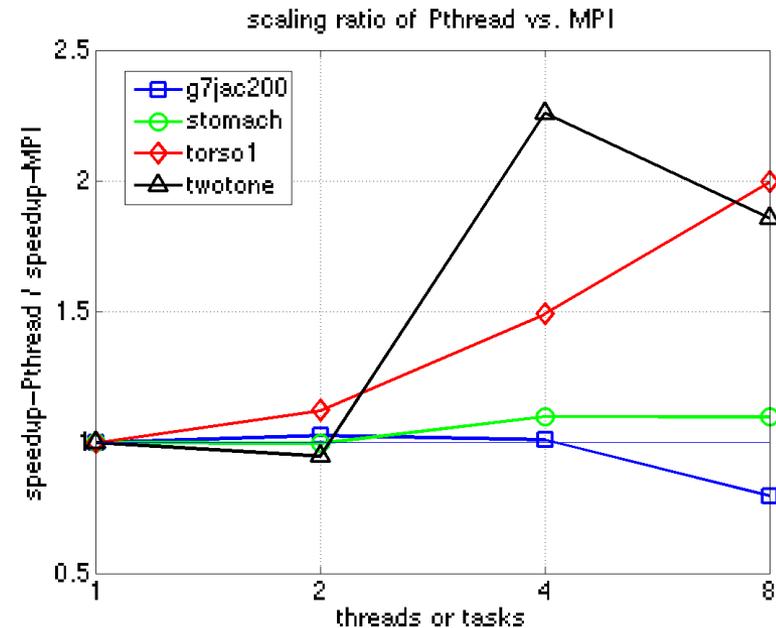
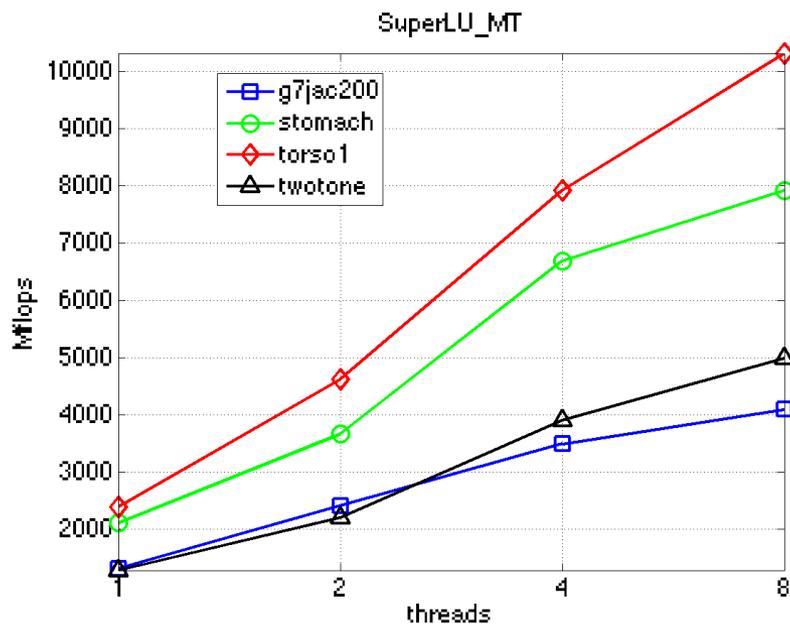
- 1.4 GHz UltraSparc T2, 1.4 Gflops/core
- 2 sockets x 8 cores/socket x **8 hardware threads/core**
- L2 cache shared: 4 MB

Benchmark matrices

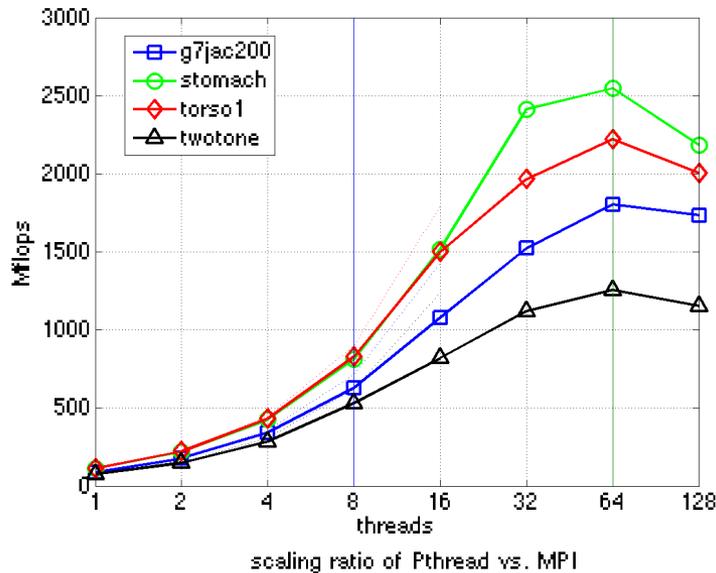


	apps	dim	nnz(A)	SLU_MT Fill	SLU_DIST Fill	Avg. S-node
g7jac200	Economic model	59,310	0.7 M	33.7 M	33.7 M	1.9
stomach	3D finite diff.	213,360	3.0 M	136.8 M	137.4 M	4.0
torso3	3D finite diff.	259,156	4.4 M	784.7 M	785.0 M	3.1
twotone	Nonlinear analog circuit	120,750	1.2 M	11.4 M	11.4 M	2.3

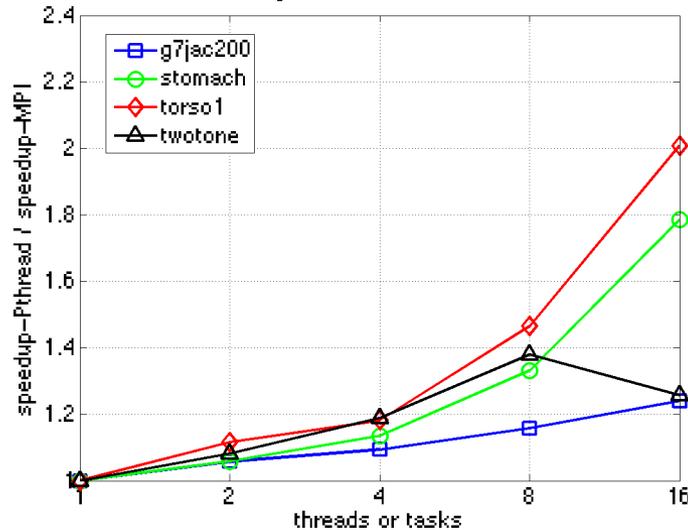
- ❖ Maximum speedup 4.3, smaller than conventional SMP
- ❖ Pthreads scale better
- ❖ Question: tools to analyze resource contention?



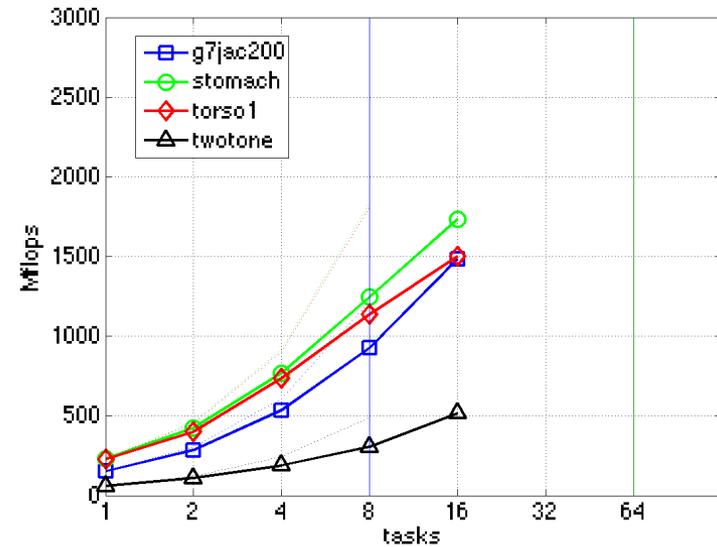
SuperLU MT



scaling ratio of Pthread vs. MPI

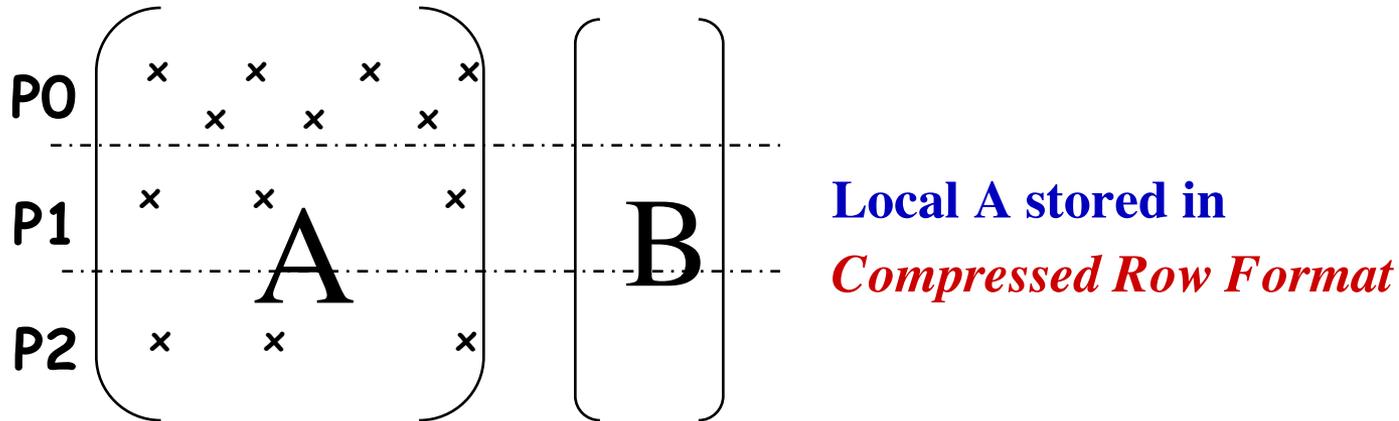


SuperLU_DIST



- Maximum speedup 20
- Pthreads more robust, scale better
- MPICH crashes with large #tasks, mismatch between coarse and fine grain models

- **Matrices involved:**
 - **A, B (turned into X)** – input, users manipulate them
 - **L, U** – output, users do not need to see them
- **A (sparse) and B (dense) are distributed by block rows**



- **Natural for users, and consistent with other popular packages: e.g. PETSc**

- Each process has a structure to store local part of A
Distributed Compressed Row Storage

```
typedef struct {  
    int_t  nnz_loc; // number of nonzeros in the local submatrix  
    int_t  m_loc;  // number of rows local to this processor  
    int_t  fst_row; // global index of the first row  
    void  *nzval;  // pointer to array of nonzero values, packed by row  
    int_t  *colind; // pointer to array of column indices of the nonzeros  
    int_t  *rowptr; // pointer to array of beginning of rows in nzval[]and colind[]  
} NRformat_loc;
```

A is distributed on 2 processors:

P0	s		u		u
	l	u			
<hr/>					
P1		l	p		
				e	u
	l	l			r

Processor P0 data structure:

- $nnz_loc = 5$
- $m_loc = 2$
- $fst_row = 0$ // **0-based indexing**
- $nzval = \{ s, u, u, | l, u \}$
- $colind = \{ 0, 2, 3, | 0, 1 \}$
- $rowptr = \{ 0, 3, 5 \}$

Processor P1 data structure:

- $nnz_loc = 7$
- $m_loc = 3$
- $fst_row = 2$ // **0-based indexing**
- $nzval = \{ l, p, | e, u, | l, l, r \}$
- $colind = \{ 1, 2, | 3, 4, | 0, 1, 4 \}$
- $rowptr = \{ 0, 2, 4, 7 \}$

- **Example: Solving a preconditioned linear system**

$$M^{-1}A x = M^{-1} b$$

$$M = \text{diag}(A_{11}, A_{22}, A_{33})$$

→ use SuperLU_DIST for each diagonal block

0	1		
2	3		
		4	5
		6	7
			8
			9
			10
			11

- **Create 3 process grids, same logical ranks (0:3), but different physical ranks**
- **Each grid has its own MPI communicator**

Two ways to create a process grid



- `superlu_gridinit(MPI_Comm Bcomm, int nrow, int ncol, gridinfo_t *grid);`
 - Maps the first $\{nrow, ncol\}$ processes in the MPI communicator Bcomm to SuperLU 2D grid
- `superlu_gridmap(MPI_Comm Bcomm, int nrow, int ncol, int usermap[], int ldumap, gridinfo_t *grid);`
 - Maps an *arbitrary* set of $\{nrow, ncol\}$ processes in the MPI communicator Bcomm to SuperLU 2D grid. The ranks of the selected MPI processes are given in usermap[] array.

For example:

	0	1	2
0	11	12	13
1	14	15	16

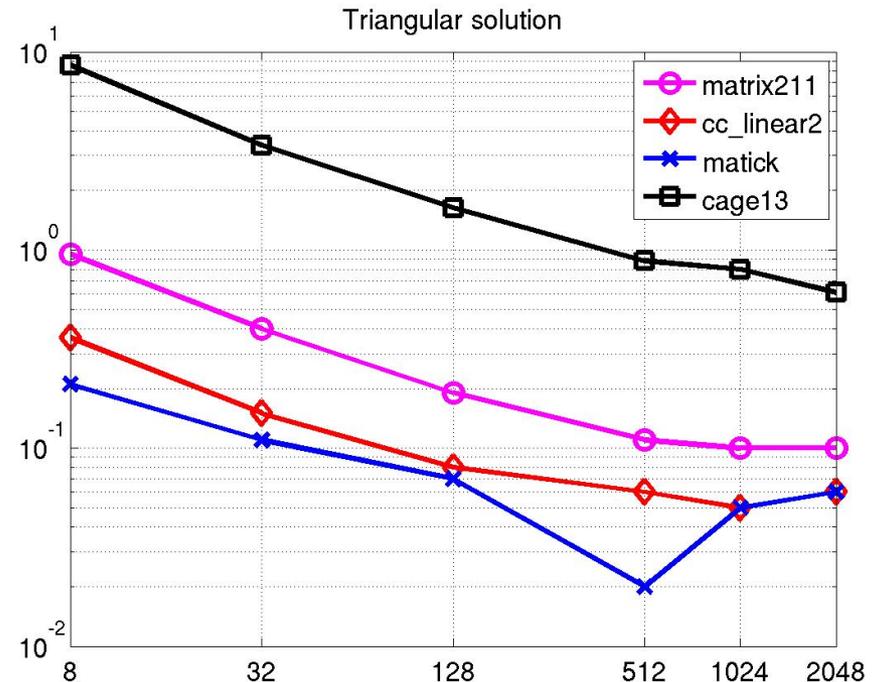
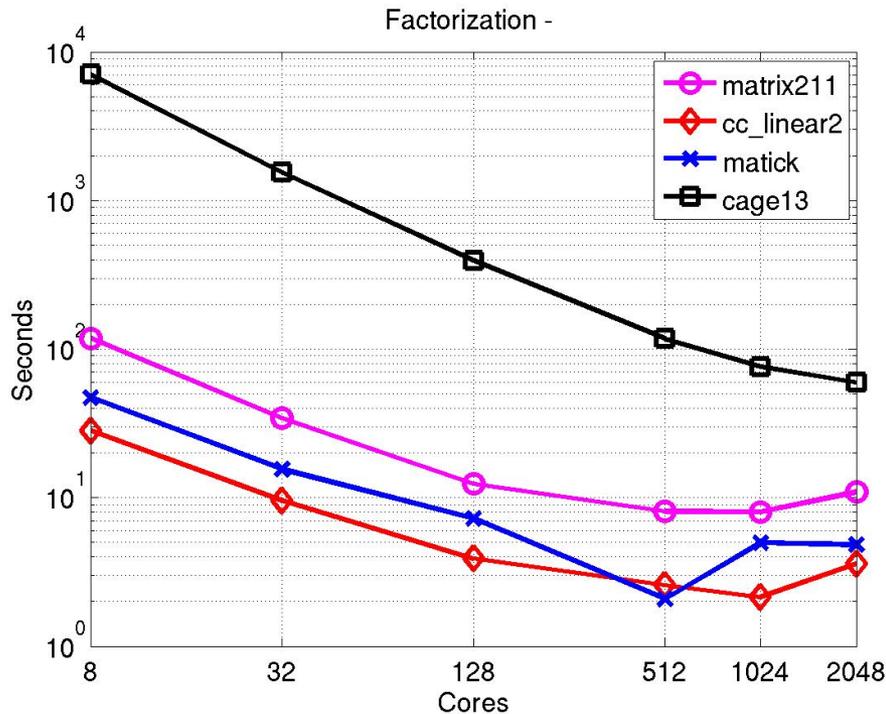
Performance of larger matrices



Name	Application	Data type	N	A / N Sparsity	L\U (10 ⁶)	Fill-ratio
matrix211	Fusion, MHD eqns (M3D-C1)	Real	801,378	161	1276.0	9.9
cc_linear2	Fusion, MHD eqns (NIMROD)	Complex	259,203	109	199.7	7.1
matick	Circuit sim. MNA method (IBM)	Complex	16,019	4005	64.3	1.0
cage13	DNA electrophoresis	Real	445,315	17	4550.9	608.5

❖ Sparsity ordering: MeTis applied to structure of $A' + A$

- #5 on the Top500 Supercomputer list
- 2 x 12-core AMD 'MagnyCours' per node, 2.1 GHz processor

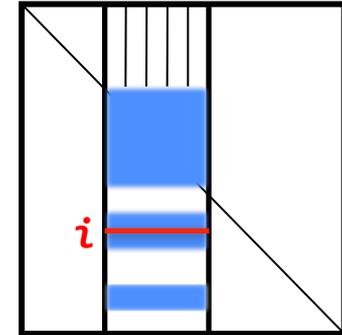


❖ Up to 1.4 Tflops factorization rate

- Available in serial SuperLU 4.0, June 2009
- Similar to ILUTP [Saad]: “T” = threshold, “P” = pivoting
 - among the most sophisticated, more robust than structure-based dropping (e.g., level-of-fill)
- ILU driver: SRC/dgsisx.c
- ILU factorization routine: SRC/dgsitr.c
- GMRES driver: EXAMPLE/ditersol.c
- Parameters:
 - `ilu_set_default_options (&options)`
 - `options.ILU_DropTol` – numerical threshold (τ)
 - `options.ILU_FillFactor` – bound on the fill-ratio (γ)

Result of Supernodal ILU (S-ILU)

- **New dropping rules S-ILU(τ , γ)**
 - supernode-based thresholding (τ)
 - adaptive strategy to meet user-desired fill-ratio upper bound (γ)



- **Performance of S-ILU**

- For 232 test matrices, S-ILU + GMRES converges with 138 cases (~60% success rate)
- S-ILU + GMRES is 1.6x faster than scalar ILU + GMRES

S-ILU for extended MHD (fusion energy)



- AMD Opteron 2.4 GHz (Cray XT5)
- ILU parameters: $\tau = 10^{-4}$, $\gamma = 10$
- Up to 9x smaller fill ratio, and 10x faster

Problems	order	Nonzeros (millions)	SuperLU		S-ILU		GMRES	
			Time	fill-ratio	time	fill-ratio	Time	Iters
matrix31	17,298	2.7 m	33.3	13.1	8.2	2.7	0.6	9
matrix41	30,258	4.7 m	111.1	17.5	18.6	2.9	1.4	11
matrix61	66,978	10.6 m	612.5	26.3	54.3	3.0	7.3	20
matrix121	263,538	42.5 m	x	x	145.2	1.7	47.8	45
matrix181	589,698	95.2 m	x	x	415.0	1.7	716.0	289

- Check sparsity ordering
- Diagonal pivoting is preferable
 - E.g., matrix is diagonally dominant, . . .
- Need good BLAS library (vendor, ATLAS, GOTO, . . .)
 - May need adjust block size for each architecture
(Parameters modifiable in routine `sp_ienv()`)
 - Larger blocks better for uniprocessor
 - Smaller blocks better for parallelism and load balance
 - Open problem: automatic tuning for block size?

- http://crd.lbl.gov/~xiaoye/SuperLU/slu_hands_on.html

- **pddrive.c**: Solve one linear system
- **pddrive1.c**: Solve the systems with same A but different right-hand side at different times
 - Reuse the factored form of A
- **pddrive2.c**: Solve the systems with the same pattern as A
 - Reuse the sparsity ordering
- **pddrive3.c**: Solve the systems with the same sparsity pattern and similar values
 - Reuse the sparsity ordering and symbolic factorization
- **pddrive4.c**: Divide the processes into two subgroups (two grids) such that each subgroup solves a linear system independently from the other.

- **SuperLU_DIST_2.5/EXAMPLE/pddrive.c**

- **Five basic steps**
 1. **Initialize the MPI environment and SuperLU process grid**
 2. **Set up the input matrices A and B**
 3. **Set the options argument (can modify the default)**
 4. **Call SuperLU routine PDGSSVX**
 5. **Release the process grid, deallocate memory, and terminate the MPI environment**

```
#include "superlu_ddefs.h"

main(int argc, char *argv[])
{
    superlu_options_t options;
    SuperLUStat_t stat;
    SuperMatrix A;
    ScalePermstruct_t ScalePermstruct;
    LUstruct_t LUstruct;
    SOLVEstruct_t SOLVEstruct;
    gridinfo_t grid;
    . . . . .
    /* Initialize MPI environment */
    MPI_Init( &argc, &argv );
    . . . . .
    /* Initialize the SuperLU process grid */
    nprow = npc = 2;
    superlu_gridinit(MPI_COMM_WORLD,
                    nprow, npc, &grid);
```

```
/* Read matrix A from file, distribute it, and set up
the right-hand side */
```

```
dcreate_matrix(&A, nrhs, &b, &ldb, &xtrue,
               &ldx, fp, &grid);
```

```
/* Set the options for the solver. Defaults are:
```

```
options.Fact = DOFACT;
options.Equil = YES;
options.ColPerm = MMD_AT_PLUS_A;
options.RowPerm = LargeDiag;
options.ReplaceTinyPivot = YES;
options.Trans = NOTRANS;
options.IterRefine = DOUBLE;
options.SolveInitialized = NO;
options.RefineInitialized = NO;
options.PrintStat = YES;
```

```
*/
set_default_options_dist(&options);
```

EXAMPLE/pddrive.c (cont.)



```
/* Initialize ScalePermstruct and LUstruct. */  
ScalePermstructInit (m, n,  
&ScalePermstruct);  
LUstructInit (m, n, &LUstruct);
```

```
/* Initialize the statistics variables. */  
PStatInit(&stat);
```

```
/* Call the linear equation solver. */
```

```
pdgssvx (&options, &A, &ScalePermstruct,  
b, ldb, nrhs, &grid, &LUstruct,  
&SOLVEstruct, berr, &stat, &info );
```

```
/* Print the statistics. */
```

```
PStatPrint (&options, &stat, &grid);
```

```
/* Deallocate storage */
```

```
PStatFree (&stat);  
Destroy_LU (n, &grid, &LUstruct);  
LUstructFree (&LUstruct);
```

```
/* Release SuperLU process grid */  
superlu_gridexit (&grid);
```

```
/* Terminate MPI execution environment */  
MPI_Finalize ();
```

```
}
```

- SuperLU_DIST_2.5/FORTRAN/
- All SuperLU objects (e.g., LU structure) are **opaque** for F90
 - They are allocated, deallocated and operated in the C side and not directly accessible from Fortran side.
- C objects are accessed via **handles** that exist in Fortran's user space
- In Fortran, all handles are of type INTEGER
- Example:

$$A = \begin{bmatrix} s & & u & u & & \\ & l & & u & & \\ & & l & p & & \\ & & & & e & u \\ l & l & & & & r \end{bmatrix}, \quad s = 19.0, u = 21.0, p = 16.0, e = 5.0, r = 18.0, l = 12.0$$

```
program f_5x5
  use superlu_mod
  include 'mpif.h'
  implicit none

  integer maxn, maxnz, maxnrhs
  parameter ( maxn = 10, maxnz = 100,
             maxnrhs = 10 )
  integer colind(maxnz), rowptr(maxn+1)
  real*8 nzval(maxnz), b(maxn),
  berr(maxnrhs)
  integer n, m, nnz, nrhs, ldb, i, ierr, info, iam
  integer nprow, npcol
  integer init
  integer nnz_loc, m_loc, fst_row
  real*8 s, u, p, e, r, l

  integer(superlu_ptr) :: grid
  integer(superlu_ptr) :: options
  integer(superlu_ptr) :: ScalePermstruct
  integer(superlu_ptr) :: LUstruct
  integer(superlu_ptr) :: SOLVEstruct
  integer(superlu_ptr) :: A
  integer(superlu_ptr) :: stat
```

! Initialize MPI environment

```
call mpi_init(ierr)
```

**! Create Fortran handles for the C structures used
! in SuperLU_DIST**

```
call f_create_gridinfo(grid)
```

```
call f_create_options(options)
```

```
call f_create_ScalePermstruct(ScalePermstruct )
```

```
call f_create_LUstruct(LUstruct)
```

```
call f_create_SOLVEstruct(SOLVEstruct)
```

```
call f_create_SuperMatrix(A)
```

```
call f_create_SuperLUStat(stat)
```

! Initialize the SuperLU_DIST process grid

```
nprow = 1
```

```
npcol = 2
```

```
call f_superlu_gridinit
```

```
(MPI_COMM_WORLD,
```

```
nprow, npcol, grid)
```

```
call get_GridInfo(grid, iam=iam)
```

f_5x5.f90 (cont.)



! Set up the input matrix A

! It is set up to use 2 processors:

! processor 1 contains the first 2 rows

! processor 2 contains the last 3 rows

```
m = 5
n = 5
nnz = 12
s = 19.0
u = 21.0
p = 16.0
e = 5.0
r = 18.0
l = 12.0
```

if (iam == 0) then

```
nnz_loc = 5
m_loc = 2
fst_row = 0      ! 0-based indexing
nzval (1) = s
colind (1) = 0   ! 0-based indexing
nzval (2) = u
colind (2) = 2
nzval (3) = u
colind (3) = 3
nzval (4) = l
colind (4) = 0
nzval (5) = u
colind (5) = 1
rowptr (1) = 0   ! 0-based indexing
rowptr (2) = 3
rowptr (3) = 5
```

else

```
nnz_loc = 7
m_loc = 3
fst_row = 2      ! 0-based indexing
nzval (1) = l
colind (1) = 1
nzval (2) = p
colind (2) = 2
nzval (3) = e
colind (3) = 3
nzval (4) = u
colind (4) = 4
nzval (5) = l
colind (5) = 0
nzval (6) = l
colind (6) = 1
nzval (7) = r
colind (7) = 4
rowptr (1) = 0   ! 0-based indexing
rowptr (2) = 2
rowptr (3) = 4
rowptr (4) = 7
```

endif

f_5x5.f90 (cont.)



```
! Create the distributed compressed row
! matrix pointed to by the F90 handle
call f_dCreate_CompRowLoc_Matrix_dist
      (A, m, n, nnz_loc, m_loc, fst_row, &
       nzval, colind, rowptr, LU_NR_loc, &
       SLU_D, SLU_GE)
```

```
! Setup the right hand side
```

```
nrhs = 1
```

```
call get_CompRowLoc_Matrix
      (A, nrow_loc=lmb)
```

```
do i = 1, lmb
  b(i) = 1.0
```

```
enddo
```

```
! Set the default input options
```

```
call f_set_default_options(options)
```

```
! Modify one or more options
```

```
Call set_superlu_options
```

```
(options, ColPerm=NATURAL)
```

```
call set_superlu_options
```

```
(options, RowPerm=NOROWPERM)
```

```
! Initialize ScalePermstruct and LUstruct
```

```
call get_SuperMatrix (A, nrow=m, ncol=n)
```

```
call f_ScalePermstructInit(m, n,
      ScalePermstruct)
```

```
call f_LUstructInit(m, n, LUstruct)
```

```
! Initialize the statistics variables
```

```
call f_PStatInit(stat)
```

```
! Call the linear equation solver
```

```
call f_pdgssvx(options, A, ScalePermstruct, b,
      lmb, nrhs, grid, LUstruct, SOLVEstruct,
      berr, stat, info)
```

```
! Deallocate the storage allocated by SuperLU_DIST
```

```
call f_PStatFree(stat)
```

```
call f_Destroy_SuperMatrix_Store_dist(A)
```

```
call f_ScalePermstructFree(ScalePermstruct)
```

```
call f_Destroy_LU(n, grid, LUstruct)
```

```
call f_LUstructFree(LUstruct)
```

f_5x5.f90 (cont.)



! Release the SuperLU process grid

call `f_superlu_gridexit(grid)`

! Deallocate the C structures pointed to by the

! Fortran handles

call `f_destroy_gridinfo(grid)`

call `f_destroy_options(options)`

Call `f_destroy_ScalePermstruct(ScalePermstruct)`

call `f_destroy_LUstruct(LUstruct)`

call `f_destroy_SOLVestruct(SOLVestruct)`

call `f_destroy_SuperMatrix(A)`

call `f_destroy_SuperLUStat(stat)`

! Terminate the MPI execution environment

call `mpi_finalize(ierr)`

Stop

end

- **Sparse LU, ILU are important kernels for science and engineering applications, used in practice on a regular basis**
- **Performance more sensitive to latency than dense case**
- **Continuing developments funded by DOE SciDAC projects**
 - **Integrate into more applications**
 - **Hybrid model of parallelism for multicore/vector nodes, differentiate intra-node and inter-node parallelism**
 - **Hybrid programming models, hybrid algorithms**
 - **Parallel HSS preconditioners**
 - **Parallel hybrid direct-iterative solver based on domain decomposition**